

A Comparison of the MODULE routines with S-PLUS Extensibility

Richard D. Langston, SAS Institute Inc.

ABSTRACT

This paper compares an extensibility feature of S-PLUS® with the capabilities of the MODULE routines supplied with Release 6.10 of the SAS® System.

INTRODUCTION

S-PLUS is a statistical and mathematical programming package that has proven to be very popular with users of the SAS System. Often S-PLUS is compared to the SAS System. The two packages use different approaches to solving problems, and these approaches cannot possibly be adequately addressed in this paper. However, one of the factors that I have heard repeated many times is that S-PLUS provides the user with the capability to call external subroutines.

With the advent of the MODULE routines in Release 6.10 of the SAS System, this capability is available in the SAS System. This paper attempts to compare this extensibility feature as it exists in S-PLUS with the functionality introduced with the MODULE routines.

The discussion in this paper will be confined to interfaces with the C language, although both S-PLUS and the SAS System can invoke FORTRAN subroutines as well as those written in other languages.

S-PLUS EXTENSIBILITY

To invoke an external C subroutine within the S-PLUS language, the S-PLUS user calls the .C object with the subroutine name and the arguments it uses. For example, consider the subroutine called *addem* that will add the first two integer arguments and set the third argument, also an integer, to the sum:

```
.C("addem",
  as.integer(x),
  as.integer(y),
  integer(1))
```

All calls the .C object must be call-by-address. Also, called routine must be of type void, meaning that it cannot return a value. Also, before the .C object can be invoked for the desired routine, the *dyn.load* routine must be invoked. (There is also a *dyn.load2* routine that may be used in certain circumstances). The loaded item can be the object file produced by the C compiler (if one is running on a UNIX® system).

Here is an example of how the user can invoke the .C object within a user-defined object:

```
Addem.compiled <-
function(x,y)
{
  if(!is.loaded(C.symbol("addem")))
    dyn.load("addem.o")
  .C("addem",
    as.integer(x),
    as.integer(y),
    integer(1)) [[3]]
}
```

Once the *Addem.compiled* object is created, it can be invoked as in the example below:

```
Addem.compiled(1,2)
```

and the S-PLUS system prints the following:

```
[1] 3
```

This indicates that the summing of 1 and 2 results in 3.

SAS SYSTEM EXTENSIBILITY

The MODULE routines allow for one to invoke external subroutines and pass arguments as needed. Although defaults can be used, the MODULE routines have the most flexibility if a special attribute table is used. This attribute table has a format as follows:

```
routine addem minarg=2 maxarg=2 returns=long;
arg 1 input required format=ib4. byvalue;
arg 2 input required format=ib4. byvalue;
```

In the above description, *addem* is a function that has two arguments, both of which are integers, since the *IB4.* format is used for each of them. The function returns a long integer.

Once the above attribute file is made available via the fileref *SASCBTBL*, the following SAS program can be run to access the *addem* function:

```
data _null_;
  x = 1; y = 2;
  z = modulen('addem',x,y);
  put z=;
run;
```

The DATA step prints the following:

```
Z=3
```

The *addem* routine must reside as an executable program. For the PC platforms, it must be an EXPORTED entry within a DLL. For the UNIX platforms (where the MODULE routines are still under development), the routine must be an executable.

SOME COMPARISONS

Now that we have seen the basic capabilities of calling external routines, let us compare and contrast the differences in the capabilities between S-PLUS and the SAS System.

Both systems use a special function to call external programs. With S-PLUS, the .C object is used. With the SAS System, it is *MODULE* if a void function is being called. However, the .C object can only call void functions; a SAS user can call functions that return scalar numeric values with the *MODULEN* function. Also, the SAS user can call functions that return scalar character arguments with the *MODULEC* function.

Both systems provide the name of the program as a quoted string. With S-PLUS, the string can refer to an executable program or to an object file that will require additional loading. The program or object will have already been loaded via a *dyn.load* or *dyn.load2*

call. With MODULE, the program must be a loadable executable program. With the first invocation of the program in a DATA or SCL step, the program will be loaded and then will remain loaded for the duration of the step. A separate load request is not performed by the user.

This has both advantages and disadvantages. Specific loading by the user allows for the program to be loaded as needed and the program stays loaded. With loading "behind the scenes" as MODULE does it, the program will be loaded and deleted in subsequent DATA and SCL steps. This can result in reduced efficiency, depending on load time. The advantage is that the user needn't be concerned about having to load the program, since it is being done automatically.

Both S-PLUS and the SAS System allow for the passing of arguments to their respective programs. S-PLUS requires that all arguments be passed by address, and it is a routine practice with S-PLUS programmers indicate the attributes for each argument passed (using *as.double*, *as.integer*, etc.). S-PLUS also allows aggregates of values to be passed as arguments, in case the called program needs to operate on vectors or matrices.

The MODULE routines also allow for arguments to be passed. These arguments can be passed by address or by value, depending on the specification in the attribute table. The default action is to pass arguments as doubles for numeric arguments, and to pass a pointer to data if character data are passed. However, the attribute table will allow for numeric data to be passed in any form that can be produced by any valid SAS format (including user-written formats). Also, the arguments can be passed by value, including character data (up to four bytes, and using the \$BYVAL format). The MODULE routines allow for aggregated arguments of any type to be placed into one or more contiguous blocks. For example, if the called program wanted to be given the address of a structure to contain a double and an integer, the MODULE routines could be directed to do this via the attribute table. The MODULE routines can also pass vectors and matrices, but that can only take place within SAS/IML® steps, where vectors and matrices are native object types.

The S-PLUS .C object is designed to load programs that were produced using the COMPILE option of the Splus command. It is possible to produce executable programs using some other compiler or some other compiler options.

The MODULE routines do not restrict the use of the compiler. As long as the arguments are passed in one of the accepted standards, the routines should be able to call the program. (An exception is for IBM mainframes, where the SAS/C compiler user is required to use the -l option.)

APPEARANCE TO THE USER

In discussing with knowledgeable S-PLUS users concerning the comparisons between the .C object and the MODULE routines, the main issue that continued to surface concerned simplicity. And it is true that with the object-oriented style of S-PLUS, it is simple to package the calls to the external programs. For example, the *Addem.compile* object defined above can simply be called with the desired arguments, as in

```
Addem.compile(1,2)
```

to cause the *addem* function to be invoked with the proper arguments and to get the proper value returned.

However, in defense of the MODULE routine example above, it could be packaged into a macro, as in

```
%macro call_add(x,y)
data _null_;
  z = modulen('addem',&x,&y);
  put z=;
run;
%mend;
```

and then be invoked by the SAS user as

```
%addem(1,2);
```

and that seems to be just as simple as the user's invocation of the *Addem.compile* object using the S-PLUS approach.

From a programming perspective, the two approaches require about the same amount of complexity. Whereas the MODULE routines obtain attribute information from an attribute file, the .C object requires the attributes to be specified within the syntax of the invocation.

USING MATRICES

The S-PLUS language is object oriented, and can treat multi-dimensional arrays as objects. These objects can be passed to subroutines to perform operations.

The DATA step language has an ARRAY statement, but in that context an array is simply a shorthand for a list of numeric or character variables. There is no assumption of the contiguity of the values in memory. And trying to use arrays with more than one dimension can be somewhat mind-boggling.

However, in SAS/IML Software, vectors and matrices are a natural part of the language. To allow for the invocation of external subroutines that will operate upon matrices, the MODULEI, MODULEIN, and MODULEIC routines are available. They are analogous to MODULE, MODULEN, and MODULEC, respectively. All of the MODULEI routines can accept scalars (1X1 matrices), vectors (1Xn matrices) as well as regular mXn matrices. For vectors and matrices, the MODULEI routines require that the values be passed by address. Scalars can be passed by value or by address. The return value can only be scalar.

It is true that if a SAS user wants to invoke an external subroutine using vectors or matrices, and wants to do so outside of the SAS/IML environment, it will be more difficult. This is not due to the lack of functionality on the part of the MODULE routines, but because both the DATA step and the SCL step invoke functions with scalar values only. (SCL does have lists, of course, but only calls its internal functions with lists as arguments).

AVOIDING ADDITIONAL C PROGRAMMING

The MODULE routines have been designed to allow the SAS user to avoid additional C programming. For this reason, the user has a large amount of flexibility in calling sequence that can be specified in the attribute table. S-PLUS is more restrictive in how the external programs can be called via the .C object, so it may be more likely that someone must write a C "wrapper" to ensure that the arguments can be passed by the .C object and make their way to the proper calling sequence of the ultimate routine.

For example, consider the *addem* routine as it is invoked by MODULE. It has two arguments, both integers, and called by value. It returns a long integer. In order for the .C object in S-SPLU to call it, the following wrapper would have to be written:

```
int addem();
void addem_wrapper(x,y,z)
int *x,*y,*z;
{
  *z = addem((int)*x,(int)*y);
}
```

Granted, this is not a complicated wrapper. And since S-PLUS does bear some resemblance to the C language, this could be an easy task for the S-PLUS programmer. If, however, the S-PLUS programmer is not familiar with C, the concept of pointers and dereferencing could be somewhat confusing.

Another example that would require a more complicated wrapper involves a function that are designed to populate a record. Consider the hypothetical function *getrec* that is given a key and will populate a memory location with a record that matches the key. The C code for invoking *getrec* would be:

```
struct RECORD {
  char name[20];
  int age;
  char sex;
  double height;
  double weight;
} record;
char key[6];
...
rc = getrec(key,&record);
...
```

After the *getrec* call, all the items in the structure are set based on the contents of the record that was obtained.

With the MODULE routines, you simply list your arguments in the attribute table as normal, and add FDSTART in the description of the first element in the structure.

The attribute table for *getrec* would be:

```
routine getrec minarg=6 maxarg=6 returns=long;
arg 1 char input format=$char6.;
arg 2 char output format=$char20. fdstart;
arg 3 num output format=ib4.;
arg 4 char output format=$char1.;
arg 5 num output format=rb8.;
arg 6 num output format=rb8.;
```

and MODULEN would be invoked as follows:

```
data _null_;
  length name $20 sex $1 height weight age 8;
  rc = modulen('getrec',name,age,sex,
              height,weight);
  put _all_;
run;
```

Note also that the MODULE routines deal properly with the padding and truncating of data. For example, if variable NAME had been length 10, MODULEN would have ensured that 20 bytes were passed to *getrec* but only 10 bytes would be copied back to NAME. Likewise, if NAME were 30 bytes, MODULEN would have ensured that NAME were blank-padded to 30 bytes properly.

With S-PLUS, a more complicated function wrapper would be necessary:

```
void getrec_wrapper(key,name,age,sex,height,
                   weight,*rc)
char *key;
```

```
char *name;
int *age;
char *sex;
double *height;
double *weight;
int *rc;
{
  struct RECORD record;

  *rc = getrec(key,&record);
  if (*rc == 0) {
    memcpy(name,record.name,20);
    *age = record.age;
    *sex = record.sex;
    *height = record.height;
    *weight = record.weight;
  }
}
```

To deal with the lengths of arguments being incorrect, extra arguments containing lengths would have to be introduced to the wrapper, and the padding or truncation would take place within the wrapper.

MISSING VALUES

The MODULE routines will convert missing values to zero if an attribute table is not supplied. If an attribute table is supplied, the conversion will be handled by the format. If the missing value makes its way to the called routine and that routine cannot handle missing values, unexpected results can occur.

S-PLUS makes more allowances for this, but this is accomplished by special code to be added to the called routine.

It is not likely that either mechanism will have great problems with this, since most external programs will not accept missing values. Part of the reason for this is that different systems and programs use different numerics for missing values.

SUMMARY

Both the S-PLUS system and the SAS System provide useful approaches to extensibility by allowing for the invoking of external programs. The S-PLUS language has integrated the capability in a more native fashion than the SAS System, but the SAS System approach does allow for more flexibility without the addition of function wrappers.

ACKNOWLEDGMENTS

I would like to thank Dr. Philip C. Spector of the University of California at Berkeley, Dr. Frank Harrell of Duke University Medical Center, and Dr. Terry M. Therneau of the Mayo Clinic for their assistance in researching this paper.

REFERENCES

SAS Institute Inc. *Microsoft Windows Environment: Changes and Enhancements to the SAS System, Release 6.11*, Cary, NC: SAS Institute Inc. Forthcoming, 1995.

SAS Institute Inc. *OS/2 Environment: Changes and Enhancements to the SAS System, Release 6.11*, Cary, NC: SAS Institute Inc. Forthcoming, 1995.

SAS Institute Inc. *Technical Support report TS-322*, Cary, NC:
SAS Institute Inc. Unpublished, but available from Technical
Support.

Statistical Sciences, Inc. *S-PLUS Programmer's Manual, Version
3.1*, Seattle: Statistical Sciences, Inc. 1993

SAS and SAS/IML are registered trademarks of SAS Institute Inc.
in the USA and other countries. ® indicates USA registration.

S-PLUS is a registered trademark of Statistical Sciences, Inc.

UNIX is a registered trademark of Novell, Inc., in the United
States and other countries; X/Open Company, Ltd., is the
exclusive licensor of such trademark.